Dijkstra

Dijkstra's algorithm is a classical approach for finding the shortest paths from a single starting node to all other nodes in a graph with non-negative edge weights. The method works by repeatedly selecting the node with the currently known shortest distance from the source and expanding its neighboring nodes, updating distances when a better route is found. This continues until all reachable nodes are processed or the destination node is reached.

In this implementation, the goal is to find the minimum-distance path from node 1 to node nnn. The solution makes use of several essential structures: an adjacency list to represent the graph efficiently, a priority queue (min-heap) to always pick the nearest unvisited node, arrays to store distances and predecessors for path reconstruction, and a visited array to avoid revisiting nodes unnecessarily. The adjacency list is particularly efficient for large graphs because it uses memory proportional to the number of edges and allows direct access to each node's neighbors.

The algorithm starts by initializing the distance to the source node as zero and all other distances as infinity. Each iteration extracts the node with the smallest tentative distance from the priority queue. For each neighbor of this node, it calculates the distance through the current node, and if this is an improvement over the previously known distance, the distance and predecessor arrays are updated, and the neighbor is added to the queue. An optimization is to stop the search as soon as the destination node is reached.

After the main loop, the algorithm checks whether a path exists by examining the distance to the target node. If no path is found, it outputs -1. Otherwise, the path is reconstructed by following predecessor pointers from the destination back to the source, then reversing the sequence to obtain the forward path.

For instance, in a graph with nodes 1 through 5 and six edges, this procedure correctly finds the shortest route 1→4→3→5 with a total weight of 5, even when alternative paths like 1→2→5 (weight 7) or 1→2→3→5 (weight 7) exist. The algorithm also properly handles multiple edges between the same nodes and self-loops, always choosing the smallest weight for relaxation.

Step-by-step summary of Dijkstra's algorithm:

Phase 1: Initialization

1. **Graph Representation** – Store all nodes and their neighbors along with edge weights.
2. **Supporting Structures** – Initialize a distance array (∞ for all except start), a predecessor array, a visited tracker, and a priority queue.

Phase 2: Processing

3. **Set Source** – Distance of the start node (node 1) is 0; push it into the queue.
4. **Loop Until Queue Empty** –

- Extract the node with the smallest distance.
- Mark it as visited.
- If it's the destination node, stop.
- For each neighbor:
    - Compute the distance through the current node.
    - If this is smaller than the previously known distance, update distance and predecessor and push neighbor into the queue.

Phase 3: Output

5. **Check for Path** – If destination distance is still ∞, no path exists.
6. **Reconstruct Path** – Follow predecessor pointers from the target to the start and reverse to get the path order.

Input:

5 6

1 2 2

2 5 5

2 3 4

1 4 1

4 3 3

3 5 1

Execution:

Step 1: Read Input

• Read n = 5, m = 6

• Create an empty graph with 6 positions (for vertices 1 to 5)

Step 2: Build Graph

• For edge 1-2 weight 2: connect vertex 1 to 2 with weight 2, and vertex 2 to 1 with weight 2

• For edge 2-5 weight 5: connect vertex 2 to 5 with weight 5, and vertex 5 to 2 with weight 5

• For edge 2-3 weight 4: connect vertex 2 to 3 with weight 4, and vertex 3 to 2 with weight 4

• For edge 1-4 weight 1: connect vertex 1 to 4 with weight 1, and vertex 4 to 1 with weight 1

• For edge 4-3 weight 3: connect vertex 4 to 3 with weight 3, and vertex 3 to 4 with weight 3

• For edge 3-5 weight 1: connect vertex 3 to 5 with weight 1, and vertex 5 to 3 with weight 1

Step 3: Initialize Data

• Distance array: vertex 1=0, vertices 2-5=infinity

• Parent array: all vertices = -1 (no parent)

• Visited array: all vertices = false (not visited)

• Priority queue: add vertex 1 with distance 0

Step 4: Process Vertex 1

• Remove vertex 1 from queue (distance 0)

• Mark vertex 1 as visited

• Check neighbor vertex 2: new distance 0+2=2, update distance to 2, set parent to 1, add to queue

• Check neighbor vertex 4: new distance 0+1=1, update distance to 1, set parent to 1, add to queue

• Queue now has: vertex 4 (distance 1), vertex 2 (distance 2)

Step 5: Process Vertex 4

• Remove vertex 4 from queue (distance 1)

• Mark vertex 4 as visited

• Check neighbor vertex 1: already visited, skip

• Check neighbor vertex 3: new distance 1+3=4, update distance to 4, set parent to 4, add to queue

• Queue now has: vertex 2 (distance 2), vertex 3 (distance 4)

Step 6: Process Vertex 2

• Remove vertex 2 from queue (distance 2)

• Mark vertex 2 as visited

• Check neighbor vertex 1: already visited, skip

• Check neighbor vertex 5: new distance 2+5=7, update distance to 7, set parent to 2, add to queue

• Check neighbor vertex 3: new distance 2+4=6, but current distance is 4, so no update

• Queue now has: vertex 3 (distance 4), vertex 5 (distance 7)

Step 7: Process Vertex 3

• Remove vertex 3 from queue (distance 4)

• Mark vertex 3 as visited

• Check neighbor vertex 2: already visited, skip

• Check neighbor vertex 4: already visited, skip

• Check neighbor vertex 5: new distance 4+1=5, better than current 7, update distance to 5, set parent to 3, add to queue

• Queue now has: vertex 5 (distance 5), vertex 5 (distance 7)

Step 8: Process Vertex 5

• Remove vertex 5 from queue (distance 5)

• Mark vertex 5 as visited

• Vertex 5 is the target, so stop the algorithm

Step 9: Check Result

• Distance to vertex 5 is 5 (not infinity), so path exists

Step 10: Reconstruct Path

• Start from vertex 5

• Parent of 5 is 3 • Parent of 3 is 4

• Parent of 4 is 1

• Path backwards: 5, 3, 4, 1

• Reverse path: 1, 4, 3, 5

Step 11: Output Result

• Print: 1 4 3 5

Output:

1 4 3 5

Pseudocode: #include <bits/stdc++.h>
using namespace std;

using ll = long long;
using pli = pair<ll,int>;

const ll INF = 1e18;

int main() {
ios::sync_with_stdio(false);
cin.tie(nullptr);

int N, M;

cin >> N >> M;

```
vector<vector<pair<int,int>>> G(N+1);
for(int i=0;i<M;i++){
    int a,b,c;
    cin >> a >> b >> c;
    G[a].emplace_back(b,c);
    G[b].emplace_back(a,c);
}

vector<ll> cost(N+1, INF);
vector<int> pred(N+1, -1);
vector<bool> visited(N+1,false);

priority_queue<pli, vector<pli>, greater<pli>> Q;
cost[1] = 0;
Q.push({0,1});

while(!Q.empty()){
    auto [d,u] = Q.top(); Q.pop();
    if(visited[u]) continue;
    visited[u] = true;
    if(u==N) break;

    for(auto &[v,w] : G[u]){
        if(!visited[v] && d+w < cost[v]){
            cost[v] = d+w;
            pred[v] = u;
```

```
            Q.push({cost[v],v});
        }
    }
}


if(cost[N]==INF){
    cout << -1 << "\n";
} else {
    vector<int> path;
    for(int x=N; x!=-1; x=pred[x]) path.push_back(x);
    reverse(path.begin(), path.end());
    for(int i=0;i<path.size();i++){
        cout << path[i] << (i+1<path.size() ? ' ' : '\n');
    }
}
}
```

Code: https://github.com/pulokdas062/Graph-Algorithm/edit/main/Dijkstra/Dijkstra%3F/Dijkstra%3F.cpp